



# Un backend per tutte le stagioni con Spring

Marcello Teodori  
[marcello.teodori@jugmilano.it](mailto:marcello.teodori@jugmilano.it)

Java User Group Milano  
<http://www.jugmilano.it>

# Due parole sul JUG Milano

- Il JUG Milano nasce il 18 Dicembre 2002 ad opera di Filippo Diotalevi e dopo un periodo di incubazione ne esce il primo meeting ufficiale il 16 Dicembre 2003
- Lo scopo è quello di favorire l'interscambio di conoscenze informatiche e creare un punto di riferimento nel panorama degli sviluppatori Java locale
- La partecipazione alle attività è del tutto gratuita e libera
- Il JUG gestisce una serie di attività tra cui meeting con cadenza regolare, scrittura di articoli, tutorial e recensioni libri sul nostro sito web, e partecipazione ad avvenimenti di rilevanza tecnologica

# Il Nostro Curriculum

- Mailing list su Yahoo! Groups con più di 400 iscritti:  
<http://tech.groups.yahoo.com/group/it-milano-java-jug/>
- wiki web site: <http://www.jugmilano.it/>
- meeting mensili con ospiti italiani ed internazionali
- Partecipazione alle principali conferenze italiane ed europee su Java, OpenSource e Metodi Agili
- Collaborazione e Promozioni con Sun, Manning, Apress ed O'Reilly
- Facciamo parte della **Top 50** dei JUG mondiali!

# Definizione di Rich Internet Application

*"web applications have extended the reach of enterprises to customers, offering anywhere and anytime access. However, this has been at the expense of the overall user-experience, which is diminished by delivery through the web browser"*

Steven Webster e Alistair McLeod, autori del libro Flex Integration with J2EE

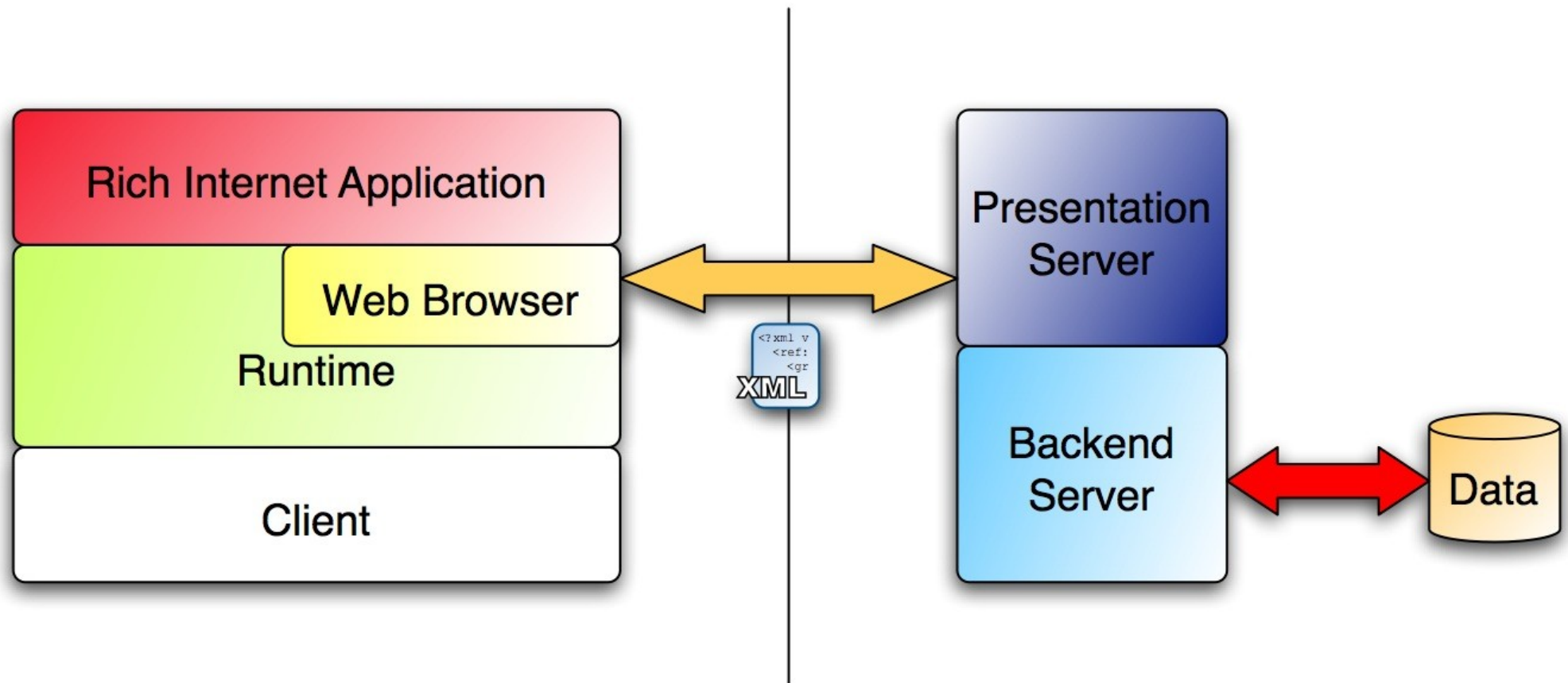
<http://www.theserverside.com/articles/article.tss?!=Flex>

=> un nuovo tipo di web application che ne superi i limiti?

Caratteristiche:

- web deployment
- logica applicativa suddivisa fra client (desktop-based) e server (browser-based)
- interazione immediata senza percezione di un roundtrip sul server
- flussi di esecuzione multipli
- accesso a rich content: audio/video

# Architettura di Riferimento



# Client Runtime

La scelta del runtime:

- JVM/JavaFX  
Il ritorno delle Applet in una veste più adatta al mondo RIA...
- AJAX  
Magia tramite JavaScript e XMLHttpRequest...
- Flash VM/Flex  
Browser plugin per animazioni vettoriali, via via arricchitosi nelle versioni recenti di funzionalità applicative.
- Silverlight  
Browser plugin che implementa una versione ridotta del runtime di .NET con funzioni GUI e media avanzate.

# Backend

Qual è il backend più adatto per le RIA?

Va bene qualsiasi soluzione, purchè esponga le proprie funzionalità in:

- XML over HTTP (ReST)
- web service “standard” SOAP
- formati “nativi” in base al runtime scelto
  - Java Serialization Protocol ed RMI over HTTP per JVM
  - AMF per Flash
  - JSON per AJAX

# Elementi di scelta per backend RIA

- Modalità di remoting già supportate dal runtime o disponibili tramite librerie aggiuntive o sviluppo custom
- Serializzazione dati nativa o parsing
- Compromesso fra effort di implementazione e performance, intesa come tempo di processing lato client e dati scambiati su rete
- Se non ho ancora deciso cosa usare sul client, le opzioni si moltiplicano ancora!!!

# Benchmarking

<http://www.jamesward.com/census/>



# Implementare Servizi con Spring

- Servizi = **POJO (Plain Old Java Object)**
- Java EE è *opzionale*
- Il core di Spring è l'**ApplicationContext** container IOC ed implementazione di **Registry e ObjectFactory**
- Configurazione tramite XML o Annotation o entrambe
- I servizi Spring sono facili da testare
- ma... uno strato servizi spring “vive” nella JVM dell'ApplicationContext, per accedervi da un'applicazione RIA devo “esportarlo”!

# Spring Remoting

- Modello generale per rendere accessibili esternamente i servizi Spring
- Adatto ad interazioni sincrone request-response
- Permette di implementare facilmente il backend di un'applicazione RIA
- Grazie alle funzionalità offerte da Spring, NON devo necessariamente pensare la mia applicazione in funzione della tecnologia con cui il client può accedere

# Spring Remoting - Exporter

- Supponiamo di aver sviluppato un'intera applicazione destinata al web (client jsp/jsf) usando Spring.
- Posso rendere accessibili i servizi offerti dai nostri bean Spring a client RIA senza modificare nulla del codice né della configurazione esistente (e funzionante), ma semplicemente **aggiungendo** opportune configurazioni.
- La chiave è il concetto di Spring Exporter
- Noi useremo principalmente questo approccio per le RIA

# Spring Remoting - Exporter

- Consente, semplicemente tramite configurazione, di rendere un servizio esistente accessibile remotamente tramite una delle tecnologie previste
- Al momento si può scegliere tra:
  - Remote Method Invocation (RMI)
  - Spring's HTTP invoker
  - Hessian (o Burlap, alternativa ad Hessian Xml based)
  - JAX-RPC, JAX-WS
  - JMS

# Spring Remoting - Exporter

- Per toccare con mano la semplicità dell'approccio, immaginiamo di avere nella nostra applicazione il seguente servizio:

```
<bean id="bookService" class="BookServiceImpl">  
    <!-- eventuali proprietà aggiuntive -->  
</bean>
```

// ed una implementazione qualsiasi.....

```
public class BookServiceImpl implements BookService {  
    public List<Book> getAllBooks() {  
        //.....  
    }  
}
```

- Nella definizione del servizio (e nell'implementazione) non c'è nulla che rimandi al remoting

# Spring Remoting - Exporter

- Immaginiamo ora di voler rendere il bean accessibile remotamente via RMI o Hessian. Tutto quello che dobbiamo fare lato server è aggiungere una delle due configurazioni seguenti:

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">  
  <property name="remoteServiceName" value="RemoteBookService"/>  
  <property name="service" ref="bookService"/>  
  <property name="serviceInterface" value="BookService"/>  
</bean>
```

```
<bean name="/RemoteBookService"  
class="org.springframework.remoting.caucho.HessianServiceExporter">  
  <property name="service" ref="accountService"/>  
  <property name="serviceInterface" value="BookService"/>  
</bean>
```

- Per utilizzare Hessian occorre naturalmente aver configurato opportunamente la DispatcherServlet

# E sul client?

- Dopo aver esposto il nostro servizio, occorre naturalmente mettere il client in grado di invocarlo. Grazie ai servizi offerti da Spring, anche questa è un'operazione banale

```
public class JavaClient {  
    private BookService bookService;  
    public void setBookService(BookService bookService) {  
        this.accountService = accountService;  
    }  
}
```

- Tramite il reference iniettato bookService, il client può invocare i metodi di BookServiceImpl (è del tutto trasparente il fatto che sia una chiamata remota)

# E sul client?

Per iniettare il riferimento corretto al JavaClient, è sufficiente aggiungere nel contesto del client una delle due configurazioni seguenti (per RMI ed Hessian)

```
<bean class="JavaClient">  
  <property name="bookService" ref="bookService"/>  
</bean>
```

```
<bean id="bookService"  
class="org.springframework.remoting.rmi.RmiProxyFactoryBean">  
  <property name="serviceUrl" value="rmi://HOST:1199/BookService"/>  
  <property name="serviceInterface" value="BookService"/>  
</bean>
```

```
<bean id="bookService"  
class="org.springframework.remoting.caucho.HessianProxyFactoryBean">  
  <property name="serviceUrl"  
    value="http://remotehost:8080/remoting/BookService"/ >  
  <property name="serviceInterface" value="BookService"/>  
</bean>
```

# Spring Remoting - Proxy

- L'approccio dell'esempio precedente, sul client sfrutta il pattern Proxy.
- Spring, per ognuna delle tecnologie elencate in precedenza, fornisce un proxy che nasconde al client tutti i dettagli della comunicazione remota.
- Pur essendo un approccio estremamente semplice e diretto, va bene solo per client java-based, perché richiede Spring sul Client
- Esistono comunque implementazioni di Spring anche per .NET, Python e ActionScript

# Spring Remoting e Java FX

- Alla luce di quanto visto, una RIA con Java FX sul client non è assolutamente complicata da integrare con i servizi remoti, essendo full java.
- Ad esempio si potrebbe scegliere la strada forse più semplice e sicura in un contesto Internet con Java su ambo i lati: HttpInvoker
- A differenza di Hessian, Spring HttpInvoker usa il meccanismo Java di serializzazione per esporre i servizi su HTTP. Questo si rivela assai vantaggioso se si vogliono scambiare tipi java complessi che non potrebbero essere serializzati via Hessian.

# Spring Remoting e Java FX

- L'uso di HttpInvoker sul server prevede l'ormai familiare configurazione:

```
<bean name="/BookService"  
class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">  
  <property name="service" ref="bookService"/>  
  <property name="serviceInterface" value="BookService"/>  
</bean>
```

- Mentre sul client potremmo utilizzare l'apposito proxy

```
<bean id="bookService"  
class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">  
  <property name="serviceUrl"  
    value="http://remotehost:8080/remoting/BookService"/>  
  <property name="serviceInterface" value="BookService"/>  
</bean>
```

# Apache CXF

- Merger dei progetti XFire e Celtix su Apache:  
<http://cxf.apache.org/>
- Implementa Spring Remoting per Web Service
- WS “standard” SOAP con JAX-WS
- WS “freeform” REST con JAX-RS
  - serializzazione XML via JAXB
  - serializzazione JSON via Jettison  
=> adatta a RIA realizzate in AJAX!

# Spring BlazeDS Integration

- Progetto congiunto SpringSource e Adobe  
<http://www.springsource.org/spring-flex>
- Implementa Spring Remoting per Flex
- Semplifica la configurazione di BlazeDS
- Usa di BlazeDS la serializzazione AMF3, tralasciando le funzionalità da “application server”  
=> la scelta migliore per applicazioni RIA realizzate con Flex

# Un esempio

Realizzazione di un semplice servizio di gestione di una libreria:

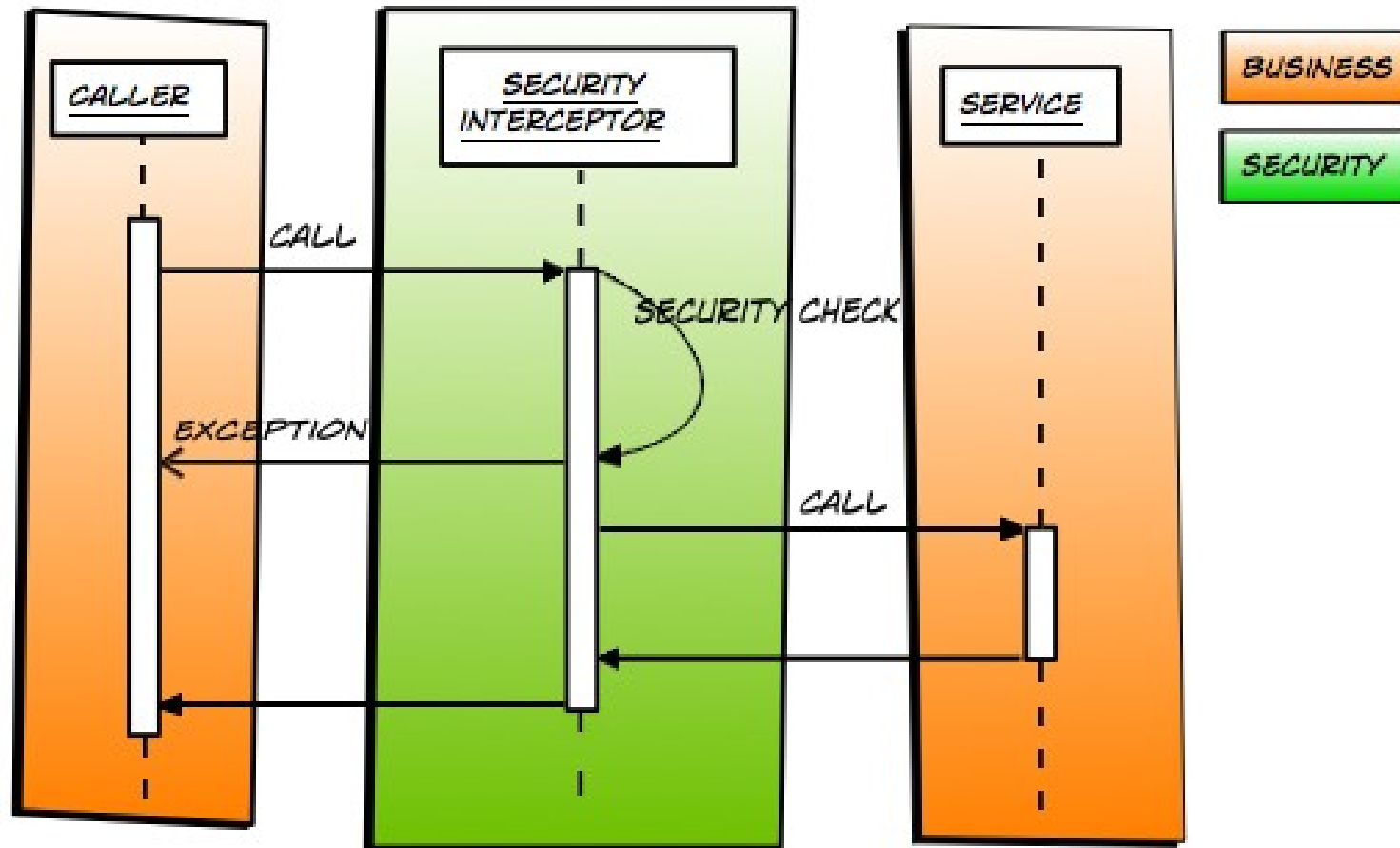
- Interfaccia LibraryService
- Implementazione LibraryServiceImpl
- Configurazione Exporter CXF su JSON  
=> per client AJAX
- Configurazione Exporter BlazeDS  
=> per client Flex

# AOP e Spring Security

- Come proteggere il nostro backend RIA?
- Integro Spring Security via AOP non modificando la mia implementazione!
- Spring Security supporta tutti i più noti meccanismi di autenticazione: su database, LDAP, NTLM e anche SSO come CAS
- Integrazione con applicazioni web “classiche” su Spring MVC tramite la DispatcherServlet, implementazione del pattern FrontController

# AOP e Spring Security

<http://www.mindtheflex.com/?p=67>



# Spring Remoting Proxy server-side

- I Proxy di Spring Remoting possono essere utilizzati anche server-side per “adattare” servizi remoti esistenti “legacy” a client RIA
- Casi d'uso:
  - integro un servizio EJB 2.1
  - proxy verso webservice non accessibile a Flex per mancanza di un file `crossdomain.xml`
  - caching di un servizio esterno molto lento a rispondere o non sempre accessibile

# Spring Remoting vs. Hibernate

- Caso d'uso: devo pubblicare un servizio Spring che accede a database tramite Hibernate
- Problema: come faccio a gestire le relazioni “lazy” nell'interazione con un client RIA senza avere la malefica `LazyInitializationException`?
- Soluzione: tramite Spring AOP serializzo solo i dati che posso effettivamente spedire e ricevere, ad esempio solo gli ID degli oggetti!

# Altri spunti...

- Spring Remoting va bene per interazioni sincrone stile request-response, e per interazioni asincrone stile publish-subscribe?
- **Spring Integration** è il nuovo progetto di Spring che definisce il modello astratto per gestirle e contiene implementazioni su JMS e thread pool
- Spring Integration è già supportato nella nuova release 1.0.0.RC1 di Spring BlazeDS Integration, seguiranno presto connettori per Jabber ecc.

# Riferimenti

- Spring Framework  
<http://www.springsource.org/>
- Cristophe Coenraets Blog – esempi su Spring BlazeDS Integration  
<http://coenraets.org/blog/>
- Flex-Mojos – plugin Maven per Flex  
<http://flexmojos.sonatype.org/>
- FNA – archetype Maven per Flex e Spring  
<http://code.google.com/p/fna/>
- CXF JSON Support  
<http://cwiki.apache.org/CXF20DOC/json-support.html>

# Q&A

